User Manual for MCmatlab

What is MCmatlab?

MCmatlab is a Monte Carlo simulation for modeling light propagation in a 3D voxel space. Fluorescence can optionally be simulated after simulation of the excitation light. Included is also a finite element simulation for temperature increase and heat diffusion in the same voxel space.

Primarily targeted for tissue optics, but can be used in any environment with turbid media in which the wave nature of light (interference phenomena) is negligible and a ray-tracing model is appropriate.

You can find the latest version of MCmatlab on https://github.com/ankrh/MCmatlab. If you publish results obtained with this software, we would be thankful if you cited its accompanying article: doi:10.1117/1.JBO.23.12.121622

License

How do I get set up?

Requirements:

- Windows 7, macOS 10.12 (Sierra) or newer or Linux
- MATLAB R2018a or newer
- (For GPU accelerated computation) A Windows PC with a CUDA-enabled graphics card and the MATLAB Parallel Computing Toolbox

Helper files:

All the helper functions needed for running MCmatlab are located in the folder (MATLAB package) "+MCmatlab", and its parent folder therefore has to be on your

MATLAB path when trying to run any MCmatlab model files. You will not need to modify any of the contained files. We recommend keeping the model files in a folder together with the "+MCmatlab" folder, such that you don't need to manually add "+MCmatlab" to your MATLAB path.

Model files:

In MCmatlab, you set up your model in a single m-file. You can find some examples to get you started in the root folder of MCmatlab. A complete list of parameters that can be set is provided below.

Media properties:

The optical properties of all media are defined in a function at the end of the model file. The examples contain some media you can modify or copy-paste to your own model file, or you define your own media from scratch in your own models. Make sure that each media in a single model file has its distinct "j"-number, as this is the number you will refer to when building your model in the geometry function (see below).

How do I use MCmatlab?

Compilation

The folders include all the executables necessary, so you don't need to compile anything. If, however, you want to change the routine in either the MCmatlab.c source code or the finiteElementHeatPropagator.c source code (both located in the folder "+MCmatlab/src"), you will need to recompile the respective mex-files. Check out those two source-files on how to do so. The source code is written in such a way that it can be compiled as either C or C++ code using either GCC, MSVC, clang or as CUDA source code using NVCC.

Building the model

You build each model in a separate m-file. Each model requires the first two and optionally more of the following steps:

1.

Build geometry (check out "Example1_StandardTissue.m", section "%% Geometry definition")
- Modify or add to "mediaPropertiesFunc" at the end of the model file to include the definitions of the media you're interested in. You may optionally include thermal and/or fluorescence properties.
- In your model file, specify the side lengths Lx, Ly, Lz of your model cuboid and the resolutions Nx, Ny, Nz.

- The "GeomFunc" you define at the end of the model file simply returns a 3D array containing the media definition for each voxel in the model cuboid. The media are referred to by a number, corresponding to the "j" defined in "mediaPropertiesFunc" at the end of the model file.
- You may optionally import your geometry from STL files as shown in example 18.
- After this step, when calling "plot(model,'G')", you will be shown a figure with the geometry you defined.

2.

Calculate light distribution (check out "Example1_StandardTissue.m", section "%% Monte Carlo simulation")
- This section in the model file contains the definitions for the Monte Carlo simulation.
- After this step, when calling "plot(model,'MC')", you will be shown various figures; one with an overview of the optical properties, one with the absorbed light in the cuboid, one with the normalized fluence rate (NFR, roughly equivalent to normalized intensity or irradiance), and one showing the fluence rate at the cuboid boundaries (taking into account only the *exiting* photon packages, not the incident beam). The last figure will only be shown if some of your cuboid boundaries are "escaping" boundaries, i.e., boundary type 1 (all boundaries are escaping boundaries) or 2 (only the top is an escaping boundary).
- (Optional) If you set "model.MC.nExamplePaths" to an integer > 0, you will also be shown a plot with the paths of the number of photons you requested.

3.

(Optional) Include Fresnel reflection and refraction
- Check out "Example3_RefractionReflection.m" and "Example17_CurvedRefractionReflection.m" on how to implement changing refractive indices in your model.

4.

(Optional) Calculate fluorescence light distribution (check out "Example5_FluorescenceAndImaging.m", section "%% Fluorescence Monte Carlo")
- To be able to run this step, your geometry definitions needs to include the fluorescence wavelength, "model.FMC.wavelength". Additionally, at least one of your media should be defined fluorescent, by defining its fluorescent yield Y in "mediaPropertiesFunc" at the end of your model file.
- After this step, when calling "plot(model,'FMC')", you will be shown the fluorescence emitter distribution, the fluorescent light fluence rate, the absorption within the cuboid, and the fluorescent light fluence rate at the cuboid boundaries.

5.

(Optional) Simulate heat distribution (check out "Example4_BloodVessel.m", section "%% Heat simulation")
- To simulate the heat diffusion, all media involved in the simulation must have the thermal properties volumetric heat capacity (VHC) and thermal conductivity (TC) defined in "mediaPropertiesFunc" at the end of your model file.
- If you want to model chemical changes such as tissue damage, the media that might undergo such change need to additionally have the Arrhenius activation energy (E) and the Arrhenius pre-exponential factor (A) defined.
- During the heat simulation, you will see an illustration of the temperature in your modelled cuboid.
- After this step, when calling "plot(model,'HS')", you will be shown various figures with the position of virtual temperature sensors in the cuboid, the temperature evolution at these positions, and whether there was some chemical change (based on the Arrhenius integral) in your cuboid.
- You can run multiple heat simulations defining different successive pulse trains as shown in "Example11_MultipleHeatSims.m"

6.

(Optional) Imaging: Model the intensity distribution received by a detector/light collector (check out ”Example5_FluorescenceAndImaging.m”, sections ”%% Monte Carlo simulation” and ”%% Fluorescence Monte Carlo”)
- Include a light collector for the incident light Monte Carlo, the fluorescence Monte Carlo, or for both. The light collector can either be a pixel array detector or a single pixel detector (such as a fibre). You can also optionally make your detector time-resolved.
- After running and calling either "plot(model,'MC')" or "plot(model,'FMC')", you will see an illustration of the geometry of your imaging system and the image of the scattered and/or fluorescence light.
- You can also choose to show the light impinging on the light collector in a time-resolved manner. Check out "Example6_TimeTagging.m" on how to do so.

7.

(Optional) Calculate the far field distribution of light escaping the simulation volume (see "Example7_FarField.m"). Note that this does not include "killed" photons.

8.

(Optional) Calculate the normalized fluence rate array of only those photons which end up on the detector (see "Example8_CollectedFluenceRate.m")

9.

(Optional) Programmatically assign values to the parameters

- See "Example9_GeometryParametricSweep.m" and "Example10_MediaPropertyParametricSweep.m" on how to implement sweeps of the parameters.

10.

(Optional) Define fluence rate-, temperature- or fractional damage-dependent optical or thermal properties (See Examples 12-15)
- By specifying the optical or thermal properties as a string (technically a char array) defining the formula as a function of fluence rate (FR), temperature (T) or fractional damage (FD), MCmatlab will take the dependence into account with an iterative approach.

11.

(Optional) Enable CUDA GPU acceleration (See Example16_CUDAacceleration.m)
- If you have an Nvidia graphics card with compute capability at least 3.0, you can set useGPU = true to enable running on the GPU, greatly speeding up both the Monte Carlo and heat simulations.

List and explanation of parameters

In the following we assume that the model object variable has been named "model". In principle, it could be given any name you want.

Geometry parameters

model.G.silentMode

[-]
(Default: False) If true, MCmatlab will not make any command window text outputs during the geometry initialization.

model.G.nx, model.G.ny, model.G.nz

[-]
The number of bins in the x, y, and z direction, respectively. Higher numbers mean a more finely resolved result, but will also require much longer runtimes to reach a low noise level.

model.G.Lx, model.G.Ly, model.G.Lz

[cm]
The side lengths of the simulation cuboid, measured in cm.

model.G.mediaPropertiesFunc

```
[-]
A MATLAB function handle. This should be set to refer to the function at the
end of the model file in which the media properties are defined. In principle
you could have several different media property functions in your model file
and the one referred to in this property is the one that MCmatlab uses, but in
practice most model files will be built using only one media property function.
```

model.G.mediaPropParams

```
[user-defined units]
A user-defined cell array that you can use to contain all sorts of inputs you
would like to use inside the mediaPropertiesFunc. This cell array is useful
when doing parametric sweeps (many MC simulations in a for or while loop) in
which the media properties vary.
```

model.G.geomFunc

```
[-]
(Similar to mediaPropertiesFunc above)
A MATLAB function handle. This should be set to refer to the function at the
end of the model file in which the model geometry is defined. In principle you
could have several different geometry functions in your model file and the one
referred to in this property is the one that MCmatlab uses, but in practice
most model files will be built using only one geometry function.
```

model.G.geomFuncParams

```
[user-defined units]
(Similar to mediaPropParams above)
A user-defined cell array that you can use to contain all sorts of inputs you
would like to use inside the geomFunc. This cell array is useful when doing
parametric sweeps (many MC simulations in a for or while loop) in which the
geometry varies.
```

(Excitation) Monte Carlo parameters

model.MC.useGPU

```
[-]
(Default: False)
If true, will run the MC simulation on GPU. If false, will use CPU.
```

model.MC.simulationTimeRequested

```
[minutes]
```

(Mutually exclusive with model.MC.nPhotonsRequested)
The time to run the MC simualtion for, in minutes. The number of photos launched will vary from run to run.

model.MC.nPhotonsRequested

[-]
(Mutually exclusive with model.MC.simulationTimeRequested)
The number of photons to launch. The execution time will vary from run to run.

model.MC.silentMode

[-]
(Default: False)
If true, MCmatlab will not make any command window text outputs during the MC simulation.

model.MC.useAllCPUs

[-]
(Default: False)
(Has no effect on MacOS or if model.MC.useGPU = true)
If true, MCmatlab will launch a number of threads equal to the number of CPU logical processors on the system. This will lead to the fastest execution but may make the system slow to respond to user inputs (mouse clicks etc.) for the duration of the simulation.
If false, MCmatlab will still be multithreaded but will leave one CPU logical processor unused. This will yield almost the same speed of execution but will make the system more responsive while the simulation is running.

model.MC.calcNFR

[-]
(Default: True)
If true, will calculate and store the normalized fluence rate (NFR) 3D array. The array takes 8*nx*ny*nz bytes of memory (and disk space, if the resulting model file is saved to disk subsequently)
For some simulations in which you are only interested, for example, in the detector image/power, you might not need the NFR and could therefore set this to false.

model.MC.calcNFRdet

[-]
(Default: False)
If true, will additionally calculate a normalized fluence rate 3D array similar to the ordinary NFR array, but only counting the areas passed through by those photons that end up registered by the detector/light collector.

model.MC.nExamplePaths

[-]
(Default: 0)
If set to a positive integer N, will store the paths of the first N photons for subsequent visualization during the plotting. This is useful to visualize how the photons move around the cuboid.

model.MC.farFieldRes

[pixels]
(Default: 0)
If set to a positive integer N, MCmatlab will calculate the far field direction angles of the escaping photons and store them in a 2D NxN array (theta,phi on the unit sphere) for later visualization.

model.MC.matchedInterfaces

[-]
(Default: True)
If true, will set all refractive indices to 1, disregarding any refractive indices specified in the media properties function. In this case, no Fresnel reflection and refraction will be simulated.

model.MC.smoothingLengthScale

[cm]
(Only used if matchedInterfaces = false)
The characteristic length scale to use for smoothing of the Sobel filter applied to the normal vectors of the non-matched interfaces within the geometry. See
https://www.spiedigitallibrary.org/journals/journal-of-biomedical-optics/volume-25/issue-02/025001/Modeling-voxel-based-Monte-Carlo-light-transport-with-curved-and/10.1117/1.JBO.25.2.025001.full
for a description of the technique used.
The use of such smoothing enables simulation of reflection and refraction even on oblique interfaces, despite the geometry being defined on a rectangular cuboid mesh. See Example17_CurvedRefractionReflection.m.

model.MC.boundaryType

[-]
0: No escaping boundaries
1: All cuboid boundaries are escaping
2: Top cuboid boundary only is escaping
Photons that hit a boundary that is "escaping" and where the refractive index is equal to 1 will be considered an "escaped" photon, and may, depending on the position and direction, be registered by the detector/light collector.

When a photon encounters a boundary that is not escaping, it is allowed to propagate beyond the edge up to a distance of 2½ times the value of (Lx or Ly or Lz) before the photon is finally killed and no longer simulated.
Allowing photons to propagate beyond the cuboid boundaries enable the photons to potentially scatter back into the simulation volume.
The more boundaries are escaping, the faster the simulation will run because fewer photons outside the cuboid have to be simulated.

model.MC.wavelength

[nm]
The wavelength of the light. This is used only to be passed on as an input to the mediaPropertiesFunc because most (if not all) optical parameters are dependent on the wavelength of the light. The wavelengths does not actually change anything in the MC simulation itself aside from the change in mua, mus and g of the involved media. The only exception is when simulation fluorescence (see below) and specifying a quantum yield fluorescer.

model.MC.beam.beamType

[-]
0: Pencil beam
A pencil beam is a beam with zero width and zero divergence.
1: Isotropically emitting line or point source
This option is the only one in which the light will originate from within the cuboid. All other options have photons packets starting at the top surface, z = 0.
2: Infinite plane wave
A beam with infinite width, but zero divergence.
3: Laguerre-Gaussian LG01 beam
A ray-based emulation of the donut-shaped Laguerre-Gaussian LG01 beam.
4: Radial-factorizable beam
A beam in which the focus beam profile can be written as I(x,y) = Ir(r), where r = sqrt(x^2+y^2), and similarly for the far field. Circular Gaussian beams can be written in this form.
5: X/Y factorizable beam
A beam in which the focus beam profile can be written as I(x,y) = Ix(x)*Iy(y), and similarly for the far field. Rectangular LED emitters can be written in this way, as well as Gaussian (circular and elliptical) beams.

model.MC.beam.emitterLength

[cm]
(Only used for model.MC.beam.beamType = 1. Default: 0)
The length of the line emitter. If zero, the emitter is a point source.

model.MC.beam.NF.radialDistr

```
[-]
(Only used for beamType = 4)
Radial near field distribution
0: Top-hat
1: Gaussian
Array: Custom. Doesn't need to be normalized.
```

model.MC.beam.NF.radialWidth

```
[cm]
(Only used for beamType = 4)
Radial near field width.
If NF.radialDistr is set to 0 or 1, this is the 1/e^2 radius
If NF.radialDistr is set to an array, this is the half-width of the full
distribution
```

model.MC.beam.NF.XDistr, model.MC.beam.NF.YDistr

```
[-]
(Only used for beamType = 5)
X and Y near field distributions
0: Top-hat
1: Gaussian
Array: Custom. Doesn't need to be normalized.
```

model.MC.beam.NF.XWidth, model.MC.beam.NF.YWidth

```
[cm]
(Only used for beamType = 5)
X and Y near field width.
If NF.XDistr/NF.YDistr is set to 0 or 1, this is the 1/e^2 radius
If NF.XDistr/Nf.YDistr is set to an array, this is the half-width of the full
distribution
```

model.MC.beam.FF.radialDistr

```
[-]
(Only used for beamType = 4)
Radial far field distribution
0: Top-hat
1: Gaussian
2: Cosine (Lambertian)
Array: Custom. Doesn't need to be normalized.
```

model.MC.beam.FF.radialWidth

```
[rad]
(Only used for beamType = 4 and if FF.radialDistr is not 2)
Radial far field width
If FF.radialDistr is set to 0 or 1, this is the 1/e^2 half-angle.
If FF.radialDistr is set to an array, this is the half-angle of the full
distribution.
For a diffraction limited Gaussian beam, this should be set to
model.MC.wavelength*1e-9/(pi*model.MC.beam.NF.radialWidth*1e-2))
```

model.MC.beam.FF.XDistr, model.MC.beam.FF.YDistr

```
[-]
(Only used for beamType = 5)
X and Y far field distributions
0: Top-hat
1: Gaussian
2: Cosine (Lambertian)
Array: Custom. Doesn't need to be normalized.
```

model.MC.beam.FF.XWidth, model.MC.beam.FF.YWidth

```
[rad]
(Only used for beamType = 5 and if FF.XDistr or FF.YDistr is not 2)
Radial far field width.
If FF.XDistr/FF.YDistr is set to 0 or 1, this is the 1/e^2 half-angle.
If FF.XDistr/FF.YDistr is set to an array, this is the half-angle of the full
distribution.
```

model.MC.beam.xFocus, model.MC.beam.yFocus model.MC.beam.zFocus

```
[cm]
The focus position x,y,z coordinates.
```

model.MC.beam.psi

```
[rad]
(Only used for beamType = 5. Default: 0)
Axial rotation angle of the beam.
```

model.MC.beam.theta, model.MC.beam.phi

```
[rad]
Polar and azimuthal angle of the beam center axis.
```

model.MC.P

```
[W]
(Only used for the thermal simulations and for any Monte Carlo simulations in
which the optical or thermal parameters depend on the fluence rate (FR),
fractional damage (FD) or temperature (T))
Incident pulse peak power.
In case of infinite plane waves, this is only the power incident upon the
cuboid's top surface.
```

model.MC.FRinitial

```
[W/cm^2]
(Only used for MC simulations in which the optical properties depend on the
fluence rate (FR))
Initial guess for the fluence rate distribution in, to be used for iterative
fluence rate dependent simulations.
```

model.MC.FRdepIterations

```
[-]
(Only used for MC simulations in which the optical properties depend on the
fluence rate (FR))
The number of iterations (times the MC simulation needs to be run) each time a
new calculation of the MC outputs is requested.
Test for yourself how high this number needs to be for your model. In my tests,
20 was enough.
```

model.MC.useLightCollector

```
[-]
(Default: False)
A logical (boolean) to specify whether the MC simulation should keep track of
whether photons hit the light collector/detector.
(The following LC properties are only used if useLightCollector = true)
```

model.MC.LC.f

```
[cm]
If the light collector is supposed to be an objective lens, this is its focal
length.
If the light collector is supposed to be a fiber tip this should be set to Inf
(infinity)
```

model.MC.LC.x, model.MC.LC.y, model.MC.LC.z

```
[cm]
(Only used if model.MC.useLightCollector = true)
```

If model.MC.LC.f is finite (light collector is an objective lens), this is the x, y, z position of the center of the focal plane of the objective lens.
If model.MC.LC.f is infinite (light collector is a fiber tip), this is the x, y, z position of the center of the fiber tip.

model.MC.LC.theta, model.MC.LC.phi

[rad]
Polar and azimuthal angle of direction that the light collector is facing

model.MC.LC.NA

[-]
(Only used for infinite f)
Fiber numerical aperture.

model.MC.LC.diam

[cm]
Diameter of the light collector aperture. For an ideal thin lens with numerical aperture NA, this is 2*f*tan(asin(NA)).

model.MC.LC.fieldSize

[cm]
(Only used for finite f)
Field size of the imaging system, that is, the diameter of area in object plane that gets imaged.

model.MC.LC.res

[pixels]
(Only used for finite f)
X and Y resolution of light collector, only used for finite f

model.MC.LC.tStart, model.MC.LC.tEnd

[s]
Start and end time of the detection time-of-flight interval. Note that photons arriving before tStart will still be registered but all binned together in a single bin containing all "too-early" photons. Likewise, all photons arriving after tEnd are binned into a "too-late" bin.

model.MC.LC.nTimeBins

[-]
(Default: 0)

Number of bins between tStart and tEnd. If zero, the measurement is not time-resolved.

Fluorescence Monte Carlo parameters

The following properties exist for fluorescence Monte Carlo simulations, and they work the same as for regular MC simulations:
FMC.useGPU, **FMC.simulationTimeRequested**, **FMC.nPhotonsRequested**, **FMC.silentMode**, **FMC.useAllCPUs**, **FMC.calcNFR**, **FMC.calcNFRdet**, **FMC.nExamplePaths**, **FMC.farFieldRes**, **FMC.matchedInterfaces**, **FMC.smoothingLengthScale**, **FMC.boundaryType**, **FMC.wavelength**, **FMC.useLightCollector**, **FMC.LC.x**, **FMC.LC.y**, **FMC.LC.z**, **FMC.LC.theta**, **FMC.LC.phi**, **FMC.LC.f**, **FMC.LC.diam**, **FMC.LC.fieldSize**, **FMC.LC.NA**, **FMC.LC.res**

Heat solver parameters

model.HS.useGPU

[-]
(Default: false)
Use CUDA acceleration for the heat simulation solver for NVIDIA GPUs

model.HS.silentMode

[-]
(Default: false)
If true, MCmatlab will not make any command window text outputs during the heat simulation.

model.HS.useAllCPUs

[-]
(Default: false)
(Has no effect on MacOS or if model.MC.useGPU = true)
If true, MCmatlab will launch a number of threads equal to the number of CPU logical processors on the system. This will lead to the fastest execution but may make the system slow to respond to user inputs (mouse clicks etc.) for the duration of the simulation.
If false, MCmatlab will launch will still be multithreaded but will leave one CPU logical processor unused. This will yield almost the same speed of execution but will make the system more responsive while the simulation is running.

model.HS.makeMovie

[-]

(Default: false)
(Only used if silentMode = false)
If true, MCmatlab will record each frame of the volumetric temperature plot and get ready to save the frames to a movie file.

model.HS.deferMovieWrite

[-]
(Default: false)
(Only used if silentMode = false and makeMovie = true)
If true, MCmatlab will save the recorded frames to a movie file. If false, the frames are still stored within the model object so that subsequent heat simulations can be concatenated to them before saving.

model.HS.largeTimeSteps

[-]
(Default: false)
If true, calculations will be faster, but some voxel temperatures may be slightly less precise. Test for yourself whether this precision is acceptable for your application.

model.HS.heatBoundaryType

[-]
The type of boundary condition to use for the heat solver.
0: Insulating boundaries
1: Constant-temperature boundaries (heat-sinked)

model.HS.nPulses

[-]
Number of consecutive pulses, each with an illumination phase and a diffusion phase. If simulating only illumination or only diffusion, use nPulses = 1.

model.HS.durationOn, model.HS.durationOff

[s]
Pulse on-duration and off-duration

model.HS.durationEnd

[s]
Non-illuminated relaxation time to add to the end of the simulation to let temperature diffuse after the pulse train.

model.HS.Tinitial

[deg C]
Initial temperature, can be scalar or 3D array of dimensions nx, ny, nz.

model.HS.plotTempLimits

[deg C]
An array of two values designating the expected range of temperatures, used only for setting the color scale in the plot.

model.HS.nUpdates

[-]
Number of times data is extracted for plots during each pulse. A minimum of 1 update is performed in each phase (2 for each pulse consisting of an illumination phase and a diffusion phase)

model.HS.mediaPropRecalcPeriod

[-]
(Used only if optical or thermal parameters have been specified to be dependent on temperature (T))
Every time this many updates have passed, the media properties will be recalculated (including, if needed, re-running MC and FMC steps). Check that this is low enough to ensure convergence of your results.

model.HS.slicePositions

[-]
(Default: [0.5 1 1])
An array of three values, describing the fractional x, y, z positions on a scale from 0 to 1 to place the volumetric slices at.

model.HS.tempSensorPositions

[cm]
(Default: empty array)
A 2D array with three columns.
Each row describes the x, y, z coordinates of a temperature sensor placed within the cuboid, at which the temperature will be recorded for subsequent plotting as a function of time.
Leave the matrix empty ([]) to disable temperature sensors.

Contribution guidelines

If you want to report a bug or are missing a feature, shoot me an email, see below.

Who do I talk to?

The main person responsible for MCmatlab is Anders Kragh Hansen: ankrh@fotonik.dtu.dk